Operating Systems - Assignment

COMP2006

Jordan Yeo
17727626

Curtin University
Science and Engineering

Contents

Introduction

The following report is for the Operating Systems Assignment for 2017. It will detail how mutual exclusion was achieved for processes and threads. Also the processes and threads that accessed shared resources. Submitted alongside this report is a README, source code and test inputs and outputs.

1 Process MSSV

For this implementation of the solution, mutual exclusion is achieved by having the parent (consumer) wait for all child (producer) processes to complete execution before continuing. This waiting is implemented by having the parent wait for the semParent semaphore before accessing the value of the shared variable representing the number of child processes finished. Once a child completes execution is waits for the semParent semaphore before acquiring a mutex lock for itself to update a shared resource, indicating it is finished. The parent is forced to wait until the value of the shared variable shows that all children are finished.

$$wait(semParent)$$
$$// \ critical \ section$$
$$signal(semParent)$$

To ensure mutual exclusion for the shared resources (buffer1, buffer2 and counter) the child waited to acquire a mutex lock before entering its critical section to modify the shared resources. Since buffer1 was never modified and only read from this did not require a mutex lock to be obtained before reading.

$$wait(semMutex)$$
$$// \ critical \ section$$
$$signal(semMutex)$$

The semaphores required (semMutex and semParent), buffer1, buffer2 and counter were implemented using shared memory. The following POSIX shared memory functions were used to create shared memory, and the corresponding functions to close the shared memory:

$$shm\_link()$$
$$ftruncate()$$
$$mmap()$$
$$shm\_unlink()$$
$$close()$$
$$munmap()$$

Zombie processes were killed with the use of signal(SIGCHILD, SIG_IGN).

2 Thread MSSV

To achieve mutual exclusion in the multi-threaded program, the parent (consumer) must wait for all threads (producer) to complete execution. The parent uses *pthread_lock_mutex()* to lock the mutex then performs a *pthread_cond_wait()* on a condition variable. The condition variable represents how many threads are still executing.

Once the thread completes execution, it acquires a mutex lock to alter the condition variable signaling it has finished its task. The thread also performs a conditional wait in the case where the parent is using the shared variable.

$$pthread\_mutex\_lock(\&mutex)$$

$$pthread\_cond\_wait(\&use, \&mutex)$$

$$pthread\_cond\_signal(\&use);$$

$$pthread\_mutex\_unlock(\&mutex)$$

In the thread MSSV the shared resources are declared as global variables. Threads have access to the variables declared global in the parent. Before a thread could access the shared resources it would first need to acquire a mutex lock. The function *pthread_lock_mutex()* blocks the caller if the mutex is in use by another. It can then alter the shared resources, counter and buffer2, before releasing the mutex lock.

To allocate the memory for buffer1, buffer2, counter and regions *malloc()* was used. The appropriate *free()* calls were used to free the allocated memory. This was done to ensure memory leaks were not present in the operation of the program.

3 Testing

3.1 Method

To test each implementation of MSSV worked as intended multiple input files were used. With these input files, multiple delays were chosen as well. Input files of various 9x9 numbers were used. Input files were used that were valid, contained one error, contained multiple errors. Also tested were smaller grids than 9x9 and empty files.

Testing was performed on the lab machines in various rooms of Building 314, Level 2.

3.2 Errors

There are no known errors in the process MSSV and the thread MSSV. Care has been taken to ensure potential memory leaks are prevented, by using the appropriate measures to free allocated memory. Memory leaks are not present in the testing of each MSSV currently performed.

3.3 Input Files

$$specTest = \begin{matrix} 6 & 2 & 4 & 5 & 3 & 9 & 1 & 8 & 7 \\ 5 & 1 & 9 & 7 & 2 & 8 & 6 & 3 & 4 \\ 8 & 3 & 7 & 6 & 1 & 4 & 2 & 9 & 5 \\ 1 & 4 & 3 & 8 & 6 & 5 & 7 & 2 & 9 \\ 9 & 5 & 8 & 2 & 4 & 7 & 3 & 6 & 1 \\ 7 & 6 & 2 & 3 & 9 & 1 & 4 & 5 & 8 \\ 3 & 7 & 1 & 9 & 5 & 6 & 8 & 4 & 2 \\ 4 & 9 & 6 & 1 & 8 & 2 & 5 & 7 & 3 \\ 2 & 8 & 5 & 4 & 7 & 3 & 9 & 1 & 6 \end{matrix}$$

$$testFail = \begin{matrix} 2 & 2 & 4 & 5 & 3 & 9 & 1 & 8 & 7 \\ 5 & 1 & 9 & 7 & 2 & 8 & 6 & 3 & 4 \\ 8 & 3 & 7 & 6 & 1 & 4 & 2 & 9 & 5 \\ 1 & 4 & 3 & 8 & 6 & 5 & 7 & 2 & 9 \\ 9 & 5 & 8 & 2 & 4 & 7 & 3 & 6 & 1 \\ 7 & 6 & 2 & 3 & 9 & 1 & 4 & 5 & 8 \\ 3 & 7 & 1 & 9 & 5 & 6 & 8 & 4 & 2 \\ 4 & 9 & 6 & 1 & 8 & 2 & 5 & 7 & 3 \\ 2 & 8 & 5 & 4 & 7 & 3 & 9 & 1 & 6 \end{matrix}$$

$$multiFail = \begin{matrix} 2 & 2 & 4 & 5 & 3 & 9 & 1 & 8 & 7 \\ 5 & 1 & 9 & 7 & 2 & 8 & 6 & 3 & 4 \\ 8 & 3 & 7 & 6 & 1 & 4 & 2 & 9 & 5 \\ 1 & 4 & 3 & 8 & 6 & 5 & 7 & 2 & 9 \\ 9 & 5 & 8 & 2 & 4 & 7 & 3 & 6 & 3 \\ 7 & 6 & 2 & 3 & 9 & 1 & 4 & 5 & 8 \\ 3 & 7 & 1 & 9 & 5 & 6 & 8 & 4 & 2 \\ 4 & 9 & 6 & 1 & 8 & 2 & 5 & 7 & 3 \\ 2 & 8 & 5 & 4 & 7 & 3 & 9 & 1 & 6 \end{matrix}$$

3.4 Expected Results

specTest

```
row 1 is valid
row 2 is valid
row 3 is valid
row 4 is valid
row 5 is valid
row 6 is valid
row 7 is valid
row 8 is valid
row 9 is valid
9 out of 9 columns are valid
9 out of 9 sub-grids are valid

There are 27 valid sub-grids, and thus the solution is valid
```

testFail

```
row 1 is invalid
row 2 is valid
row 3 is valid
row 4 is valid
row 5 is valid
row 6 is valid
row 7 is valid
row 8 is valid
row 9 is valid
8 out of 9 columns are valid
8 out of 9 sub-grids are valid

There are 24 valid sub-grids, and thus the solution is invalid
```

multiFail

```
row 1 is invalid
row 2 is valid
row 3 is valid
row 4 is valid
row 5 is invalid
row 6 is valid
row 7 is valid
row 8 is valid
row 9 is valid
7 out of 9 columns are valid
7 out of 9 sub-grids are valid

There are 21 valid sub-grids, and thus the solution is invalid
```

3.5 Actual Results

Figure 1: Processes: specTest

```
[17727626@lab221-c01 partA]$ ./mssv ../testFiles/specTest.txt 10
Validation result from process ID-19029: row 1 is valid
Validation result from process ID-19030: row 2 is valid
Validation result from process ID-19031: row 3 is valid
Validation result from process ID-19032: row 4 is valid
Validation result from process ID-19033: row 5 is valid
Validation result from process ID-19034: row 6 is valid
Validation result from process ID-19035: row 7 is valid
Validation result from process ID-19036: row 8 is valid
Validation result from process ID-19037: row 9 is valid
Validation result from process ID-19038: 9 out of 9 columns are valid
Validation result from process ID-19039: 9 out of 9 sub-grids are valid
There are 27 valid sub-grids, and thus the solution is valid
```

Figure 2: Threads: specTest

```
[17727626@lab221-b01 partB]$ ./mssv ../testFiles/specTest.txt 10
Validation result from thread ID-2979485440: row 1 is valid
Validation result from thread ID-2971092736: row 2 is valid
Validation result from thread ID-2962700032: row 3 is valid
Validation result from thread ID-2954307328: row 4 is valid
Validation result from thread ID-2945914624: row 5 is valid
Validation result from thread ID-2937521920: row 6 is valid
Validation result from thread ID-2929129216: row 7 is valid
Validation result from thread ID-2920736512: row 8 is valid
Validation result from thread ID-2912343808: row 9 is valid
Validation result from thread ID-2903951104: 9 out of 9 columns are valid
Validation result from thread ID-2895558400: 9 out of 9 sub-grids are valid
There are 27 valid sub-grids, and thus the solution is valid
```

Figure 3: Processes: testFail

```
[17727626@lab221-c01 partA]$ ./mssv ../testFiles/testFail.txt 10
Validation result from process ID-19053: row 1 is invalid
Validation result from process ID-19054: row 2 is valid
Validation result from process ID-19055: row 3 is valid
Validation result from process ID-19056: row 4 is valid
Validation result from process ID-19057: row 5 is valid
Validation result from process ID-19058: row 6 is valid
Validation result from process ID-19059: row 7 is valid
Validation result from process ID-19060: row 8 is valid
Validation result from process ID-19061: row 9 is valid
Validation result from process ID-19062: 8 out of 9 columns are valid
Validation result from process ID-19063: 8 out of 9 sub-grids are valid
There are 24 valid sub-grids, and thus the solution is invalid
```

Figure 4: Threads: testFail

```
[17727626@lab221-b01 partB]$ ./mssv ../testFiles/testFail.txt 10
Validation result from thread ID-2959283968: row 1 is invalid
Validation result from thread ID-2950891264: row 2 is valid
Validation result from thread ID-2942498560: row 3 is valid
Validation result from thread ID-2934105856: row 4 is valid
Validation result from thread ID-2925713152: row 5 is valid
Validation result from thread ID-2917320448: row 6 is valid
Validation result from thread ID-2908927744: row 7 is valid
Validation result from thread ID-2900535040: row 8 is valid
Validation result from thread ID-2892142336: row 9 is valid
Validation result from thread ID-2883749632: 8 out of 9 columns are valid
Validation result from thread ID-2875356928: 8 out of 9 sub-grids are valid
There are 24 valid sub-grids, and thus the solution is invalid
```

Figure 5: Processes: multiFail

```
[17727626@lab221-c01 partA]$ ./mssv ../testFiles/multiFail.txt 10
Validation result from process ID-19070: row 1 is invalid
Validation result from process ID-19071: row 2 is valid
Validation result from process ID-19072: row 3 is valid
Validation result from process ID-19073: row 4 is valid
Validation result from process ID-19074: row 5 is invalid
Validation result from process ID-19075: row 6 is valid
Validation result from process ID-19076: row 7 is valid
Validation result from process ID-19077: row 8 is valid
Validation result from process ID-19078: row 9 is valid
Validation result from process ID-19079: 7 out of 9 columns are valid
Validation result from process ID-19080: 7 out of 9 sub-grids are valid
There are 21 valid sub-grids, and thus the solution is invalid
```

Figure 6: Threads: multiFail

```
[17727626@lab221-b01 partB]$ ./mssv ../testFiles/multiFail.txt 10
Validation result from thread ID-3998418688: row 1 is invalid
Validation result from thread ID-3990025984: row 2 is valid
Validation result from thread ID-3981633280: row 3 is valid
Validation result from thread ID-3973240576: row 4 is valid
Validation result from thread ID-3964847872: row 5 is invalid
Validation result from thread ID-3956455168: row 6 is valid
Validation result from thread ID-3990025984: row 7 is valid
Validation result from thread ID-3813844736: row 8 is valid
Validation result from thread ID-3948062464: row 9 is valid
Validation result from thread ID-3939669760: 7 out of 9 columns are valid
Validation result from thread ID-3931277056: 7 out of 9 sub-grids are valid
There are 21 valid sub-grids, and thus the solution is invalid
```

4 README

4.1 Purpose

The program validates an input file containing a sudoku solution. There are two versions. One utilising processes and the other utilising threads.

4.2 Running the Program

To compile the C files into an executable format.

```
make
```

To run the program there are two options.

Option 1:

```
make run
```

This will only let you run with the preset parameters and they will need to be altered in the Makefile to test other *input files* and *delays*

```
INPUT: ../testFiles/specTest.txt
DELAY: 10
```

Option 2:

```
./mssv <inputFile> <delay>
```

Between each time the program is run, the following command should be entered and executed. This is to delete the logfile produced from an invalid test file.

```
make clean
```

4.3 Files

partA

- Makefile
- mssv.c
- mssv.h

partB

- Makefile
- mssv.c
- mssv.h

testFiles

- test files

References

[1] *Interprocess communication using POSIX Shared Memory in Linux*. URL: https://www.softprayog.in/programming/interprocess-communication-using-posix-shared-memory-in-linux.

[2] *POSIX Semaphores*. URL: https://www.softprayog.in/programming/posix-semaphores.

[3] *POSIX Threads Programming in C*. URL: https://www.softprayog.in/programming/posix-threads-programming-in-c.

[4] *POSIX Threads Synchronization in C*. URL: https://www.softprayog.in/programming/posix-threads-synchronization-in-c.

5 Appendices

5.1 Processes: mssv.c

```c
#include "mssv.h"

int main (int argc, char* argv[])
{
    // Validate command line parameters
    validateUse(argc, argv);

    // Rename command line parameters
    char* inputFile = argv[1];
    int maxDelay = atoi(argv[2]);

    // Variables
    int i, pid, processNum, numbers[] = {0,0,0,0,0,0,0,0,0};
    Region* region;
    sem_t semMutex, semParent, *semaphores;

    // File Descriptors
    int buff1FD, buff2FD, counterFD, semFD, regionFD, resFD;

    // Shared memory pointers
    int *buff2Ptr, *countPtr, *resourceCount, (*buff1Ptr)[NINE][NINE];

    // Generate random maxDelay
    srand((unsigned) time(NULL));

    maxDelay = rand() % maxDelay;

    // Create shared memory
    initMemory( &buff1FD, &buff2FD, &counterFD, &semFD, &regionFD, &resFD);

    // Map shared memory to pointers
    mapMemory(&buff1FD, &buff2FD, &counterFD, &semFD, &regionFD, &resFD,
                &buff1Ptr, &buff2Ptr, &countPtr, &semaphores, &region,
                    &resourceCount);

    // Initialise semaphores
    if ((sem_init(&semMutex, 1, 1)== 1) ||(sem_init(&semParent, 1, 1) == 1))
    {
        fprintf(stderr, "Could not initialise semaphores\n");
        exit(1);
    }

    semaphores[0] = semMutex;
    semaphores[1] = semParent;

    // Initialise parameters
    *countPtr = 0;
    pid = -1;
    processNum = 0;

    // Read input file
    readFile(inputFile, NINE, NINE, buff1Ptr);

    // Parent aquires lock of resourceCount
    sem_wait(&(semaphores[1])); // Lock child

    *resourceCount = 0;

    // Create child processes for
    while( processNum < NUMPROCESSES && pid != 0 )
    {
        signal(SIGCHLD, SIG_IGN); // Kill zombie processNum-1
        pid = fork();
```

```
64
65          // Allow the parent to increment shared variable count
66          if ( pid > 0)
67          {
68              *resourceCount = *resourceCount + 1;
69          }
70          processNum++;
71      }
72
73      if( pid == 0) // Child process
74      {
75          childManager(region, semaphores, buff1Ptr, buff2Ptr, countPtr,
76                               resourceCount,  processNum, numbers, maxDelay );
77      }
78      else if ( pid > 0) // Parent process
79      {
80          parentManager(region, semaphores, countPtr, resourceCount);
81
82          // Clean up shared memory
83          cleanMemory(&buff1Ptr, &buff2Ptr, &countPtr, &semaphores,
84                          &region, &resourceCount, buff1FD, buff2FD, counterFD,
85                               semFD, regionFD, resFD);
86      }
87      else // Unsuccessful child process creation attempt
88      {
89          fprintf(stderr, "Unable to create child processes. Please run \"killall mssv\"\n");
90      }
91  }
92
93  /*****************************************************************************/
94
95  /**
96   * Read the contents of the input file passed as a command line argument
97   * @param inputFile File to be read
98   * @param rows      Number of rows in matrix
99   * @param cols      Number of columns in matrix
100  * @param buffer    Matrix to store contents of input file
101  */
102 void readFile(char* inputFile, int rows, int cols, int (*buffer)[rows][cols])
103 {
104     FILE* inStrm;
105     int i, j;
106
107     inStrm = fopen(inputFile, "r"); // Open file for reading
108
109     if (inStrm == NULL) // Check file opened correctly
110     {
111         perror("Error opening file for reading\n");
112         exit(1);
113     }
114
115     // Store contents of file in 2D array
116     for( i = 0; i < rows; i++ )
117     {
118         for ( j = 0; j < cols; j++ )
119         {
120             fscanf( inStrm, "%d", &(*(buffer))[i][j] );
121         }
122     }
123
124     fclose(inStrm); // Close file
125 }
126
127 /**
128  * Write the invalid regions to log file
129  * @param region Sub region
130  * @param format String to be written
131  */
```

```
132 void writeFile(Region* region, char* format)
133 {
134     char* filename = "logfile";
135     FILE* outFile;
136     int val;
137 
138     outFile = fopen(filename, "a"); // Open file for appending
139     if (outFile == NULL) // Check file opened correctly
140     {
141         perror("Error opening file for writing\n");
142         exit(1);
143     }
144 
145     fprintf(outFile, "process ID-%d: %s",region->pid, format);
146 
147     fclose(outFile); // Close file
148 }
149 
150 /**
151  * Set each index to zero
152  * @param numbers Array to be reset
153  */
154 void resetArray(int numbers[])
155 {
156     for (int i = 0; i < NINE; i++)
157     {
158         numbers[i] = 0;
159     }
160 }
161 
162 /**
163  * Check if the contents of the array has any value other than one
164  * @param  numbers Array to be checked
165  * @return         Status of array being valid or not
166  */
167 int checkValid(int numbers[])
168 {
169     for (int j = 0; j < NINE; j++)
170     {
171         if ( numbers[j] != 1)
172         {
173             return FALSE;
174         }
175     }
176 
177     return TRUE;
178 }
179 
180 /**
181  * Handles the routine for the parent process. Outputs the result to the screen
182  * @param region        Array containing each region struct
183  * @param semaphores    Array of all semaphores
184  * @param countPtr      Pointer to shared memory counter
185  * @param resourceCount Status of number of child processes executing
186  */
187 void parentManager(Region *region, sem_t *semaphores, int* countPtr,
188                    int* resourceCount)
189 {
190     char *type, *message;
191     sem_post(&(semaphores[1])); // Unlock child
192     int done = FALSE;
193     int position;
194 
195     while( !done ) // Wait for all children to finish executing
196     {
197         //printf("Parent Waiting for Children\n");
198         sem_wait(&(semaphores[1])); // Lock child
199         if ( *resourceCount == 0)
```

```
200         {
201             done = TRUE;
202         }
203         sem_post(&(semaphores[1])); // Unlock child
204 
205     }
206 
207     for(int ii = 0; ii < NUMPROCESSES; ii++)
208     {
209         sem_wait(&(semaphores[0])); //Lock mutex
210         if (region[ii].type == ROW)
211         {
212             position = region[ii].positionX;
213             type = "invalid";
214 
215             if (region[ii].valid == TRUE)
216             {
217                 type = "valid";
218             }
219             printf("Validation result from process ID-%d: row %d is %s\n",
220                                     region[ii].pid, position, type);
221         }
222         else if (region[ii].type == COL)
223         {
224             type = "column";
225             position = region[ii].positionX;
226             printf("Validation result from process ID-%d: %d out of 9 columns are valid\n"
227                         ,region[ii].pid, region[ii].positionX);
228         }
229         else
230         {
231             type = "sub-grid";
232             position = region[ii].positionX;
233 
234             printf("Validation result from process ID-%d: %d out of 9 sub-grids are valid\n"
235                         ,region[ii].pid, region[ii].positionX);
236 
237         }
238 
239         sem_post(&(semaphores[0])); //Unlock mutex
240     }
241 
242 
243   if (*countPtr == 27)
244   {
245       message = "valid";
246   }
247   else
248   {
249       message = "invalid";
250   }
251 
252   printf("There are %d valid sub-grids, and thus the solution is %s\n", *countPtr, message);
253 }
254 
255 
256 /**
257  * Routine for child processes. Check the validity of sub region.
258  * @param region        Sub-region struct for each process
259  * @param semaphores    Array of semaphores
260  * @param buff1Ptr      Pointer to buffer1 in shared memory
261  * @param buff2Ptr      Pointer to buffer2 in shared memory
262  * @param countPtr      Pointer to counter in shared memory
263  * @param resourceCount Pointer to resourceCount in shared memory
264  * @param processNum    Child process number
265  * @param numbers       Array of numbers to check validity of sub region
266  * @param maxDelay      Delay for each process
267  */
```

```
268 void childManager(Region *region, sem_t *semaphores, int (*buff1Ptr)[NINE][NINE],
269                     int *buff2Ptr, int* countPtr, int* resourceCount,
270                         int processNum, int *numbers, int maxDelay )
271 {
272     char format[500];
273     int numValid, comma = 0;
274 
275     if( processNum <= 9) // Check a row in buffer1
276     {
277         for (int i = 0; i < NINE; i++)
278         {
279             // Update numbers array
280             numbers[((*buff1Ptr)[processNum-1][i])-1]++;
281         }
282 
283         sleep(maxDelay); // Sleep
284         sem_wait(&(semaphores[0])); //Lock mutex
285 
286         // Update region struct
287         region[processNum-1].type = ROW;
288         region[processNum-1].positionX = processNum;
289         region[processNum-1].pid = getpid();
290         region[processNum-1].valid = checkValid(numbers);
291 
292         numValid = 0;
293         if (region[processNum-1].valid == TRUE)
294         {
295             numValid = 1;
296         }
297         else // Write to log file
298         {
299             sprintf(format, "row %d is invalid\n", processNum);
300             writeFile(&(region[processNum-1]), format);
301         }
302 
303         buff2Ptr[processNum-1] = numValid; // Update buffer2
304 
305         *countPtr = *countPtr + numValid; // Update counter
306 
307         sem_post(&(semaphores[0])); // Unlock child
308 
309     }
310     else if(processNum == 10) // Check all columns
311     {
312         sprintf(format, "column ");
313 
314         int validCol = 0;
315         for ( int nn = 0; nn < NINE; nn++) // Iterate through each column
316         {
317             for(int ii = 0; ii < NINE; ii++) // Iterate through each row
318             {
319                 numbers[(*buff1Ptr)[ii][nn]-1]++; // Update numbers array
320             }
321 
322             if ( checkValid( numbers) == TRUE )
323             {
324               validCol++;
325             }
326             else
327             {
328                 if (comma == 0)
329                 {
330                     comma = 1;
331                     sprintf(format + strlen(format), "%d", nn+1);
332                 }
333                 else
334                 {
335                     sprintf(format + strlen(format), ", %d ", nn+1);
```

```
336                 }
337             }
338 
339             resetArray(numbers);
340         }
341 
342         sleep(maxDelay);
343         if (validCol == 8)
344         {
345             sprintf(format + strlen(format), " is invalid\n");
346         }
347         else
348         {
349             sprintf(format + strlen(format), "are invalid\n");
350         }
351         sem_wait(&(semaphores[0])); //Lock mutex
352 
353         // Update region struct
354         region[processNum-1].type = COL;
355         region[processNum-1].positionX = validCol;
356         region[processNum-1].pid = getpid();
357         if(validCol != 9)
358         {
359             writeFile(&(region[processNum-1]), format);
360         }
361 
362         numValid = region[processNum-1].positionX;
363 
364         buff2Ptr[processNum-1] = validCol; // Update buffer2
365 
366         *countPtr = *countPtr + validCol; // Update counter
367 
368         sem_post(&(semaphores[0])); // Unlock mutex
369     }
370     else if( processNum == 11) // Check sub-grids
371     {
372         sprintf(format, "sub-grid ");
373 
374         int validSub = 0;
375 
376         // Iterate through each of the 9 3x3 sub-grid
377         for ( int jj = 0; jj < 3; jj++)
378         {
379             for (int kk = 0; kk < 3; kk++)
380             {
381                 for (int ll = jj*3; ll < jj*3+3; ll++)
382                 {
383                     for (int mm = kk*3; mm < kk*3+3; mm++)
384                     {
385                         // Update numbers array
386                         numbers[(*buff1Ptr)[ll][mm]-1]++;
387                     }
388               }
389 
390               if ( checkValid(numbers) == TRUE )
391               {
392                     validSub++;
393               }
394               else // Update string for log file
395               {
396                   if (comma == 0)
397                   {
398                       comma = 1;
399                       sprintf(format+strlen(format), "[%d..%d, %d..%d]",
400                                   jj*3+1, jj*3+3, kk*3+1, kk*3+3);
401                   }
402                   else
403                   {
```

```
404                         sprintf(format+strlen(format), ", [%d..%d, %d..%d] ",
405                                 jj*3+1, jj*3+3, kk*3+1, kk*3+3);
406                     }
407                 }
408                 resetArray(numbers);
409             }
410 
411         }
412 
413         sleep(maxDelay);
414         if (validSub == 8)
415         {
416             sprintf(format+strlen(format), " is invalid\n");
417         }
418         else
419         {
420             sprintf(format+strlen(format), "are invalid\n");
421         }
422         sem_wait(&(semaphores[0])); //Lock mutex
423 
424         // Update region struct
425         region[processNum-1].type = SUB_REGION;
426         region[processNum-1].positionX = validSub;
427         region[processNum-1].pid = getpid();
428 
429         if(validSub != 9) // Write to log file
430         {
431             writeFile(&(region[processNum-1]), format);
432         }
433 
434         buff2Ptr[processNum-1] = validSub; // Update buffer 2
435 
436         *countPtr = *countPtr + validSub; // Update counter
437 
438         sem_post(&(semaphores[0])); // Unlock mutex
439     }
440 
441     // Child signals it is finished by incremented resourceCount
442     sem_wait(&(semaphores[1])); // Lock child
443         *resourceCount = *resourceCount - 1;
444     sem_post(&(semaphores[1])); // Unlock child
445 
446 }
447 
448 /**
449  * Initalise shared memory constructs
450  * @param buff1FD   File descriptor for buffer1
451  * @param buff2FD   File descriptor for buffer2
452  * @param counterFD File descriptor for counter
453  * @param semFD     File descriptor for semaphores
454  * @param regionFD  File descriptor for regions
455  * @param resFD     File descriptor for resourceCount
456  */
457 void initMemory( int* buff1FD, int* buff2FD, int* counterFD, int* semFD,
458                  int* regionFD, int* resFD)
459 {
460 
461     // Create shared memory
462     *buff1FD = shm_open("buffer1", O_CREAT | O_RDWR, 0666);
463     *buff2FD = shm_open("buffer2", O_CREAT | O_RDWR, 0666);
464     *counterFD = shm_open("counter", O_CREAT | O_RDWR, 0666);
465     *semFD = shm_open("semaphores", O_CREAT | O_RDWR, 0666);
466     *regionFD = shm_open("region", O_CREAT | O_RDWR, 0666);
467     *resFD = shm_open("resources", O_CREAT | O_RDWR, 0666);
468 
469     // Check shared memory was created correctly
470     if ( *buff1FD == -1 || *buff2FD == -1 || *counterFD == -1 || *semFD == -1 ||
471             *regionFD == -1 || *resFD == -1 )
```

```
472     {
473         fprintf( stderr, "Error creating shared memory blocks\n" );
474         exit(1);
475     }
476 
477     // Set size of shared memory constructs
478     if ( ftruncate(*buff1FD, sizeof(int) * NINE * NINE) == -1 )
479     {
480         fprintf( stderr, "Error setting size of buffer1" );
481         exit(1);
482     }
483 
484     if ( ftruncate(*buff2FD, sizeof(int) * NUMPROCESSES) == -1 )
485     {
486         fprintf( stderr, "Error setting size of buffer2" );
487         exit(1);
488     }
489 
490     if ( ftruncate(*counterFD, sizeof(int)) == -1 )
491     {
492         fprintf( stderr, "Error setting size of counter" );
493         exit(1);
494     }
495 
496     if ( ftruncate(*semFD, sizeof(sem_t) * 2 ) == -1 )
497     {
498         fprintf( stderr, "Error setting size of semaphores" );
499         exit(1);
500     }
501 
502     if ( ftruncate(*regionFD, sizeof(Region)*NUMPROCESSES) == -1 )
503     {
504         fprintf( stderr, "Error setting size of regions" );
505         exit(1);
506     }
507 
508     if ( ftruncate(*resFD, sizeof(int)) == -1 )
509     {
510         fprintf( stderr, "Error setting size of resourceCount" );
511         exit(1);
512     }
513 }
514 
515 /**
516  * Map shared memory to addresses
517  * @param buff1FD       File descriptor for buffer1
518  * @param buff2FD       File descriptor for buffer2
519  * @param counterFD     File descriptor for counter
520  * @param semFD         File descriptor for semaphores
521  * @param regionFD      File descriptor for regions
522  * @param resFD         File descriptor for resourceCount
523  * @param buff1Ptr      Pointer to buffer1 in shared memory
524  * @param buff2Ptr      Pointer to buffer2 in shared memory
525  * @param countPtr      Pointer to counter in shared memory
526  * @param semaphores    Array of semaphores
527  * @param region        Array of region structs
528  * @param resourceCount Pointer to resourceCount in shared memory
529  */
530 void mapMemory(int* buff1FD, int* buff2FD, int* counterFD, int* semFD,
531                    int* regionFD, int* resFD, int (**buff1Ptr)[NINE][NINE], int (**buff2Ptr),
532                         int** countPtr, sem_t** semaphores, Region** region, int**
    resourceCount)
533 {
534     // Memory mapping
535     *buff2Ptr = (int*) mmap(NULL, sizeof(int)*NINE*NINE,
536                 PROT_READ | PROT_WRITE, MAP_SHARED, *buff2FD, 0);
537     *buff1Ptr = mmap(NULL, sizeof(int)*NUMPROCESSES,
538                 PROT_READ | PROT_WRITE, MAP_SHARED, *buff1FD, 0);
```

```
539     *countPtr = (int*) mmap(NULL, sizeof(int),
540                 PROT_READ | PROT_WRITE, MAP_SHARED, *counterFD, 0);
541     *semaphores = mmap(NULL, sizeof(sem_t) * 2,
542                 PROT_READ | PROT_WRITE, MAP_SHARED, *semFD, 0);
543     *region = mmap(NULL, sizeof(Region)*NUMPROCESSES,
544                 PROT_READ | PROT_WRITE, MAP_SHARED, *regionFD, 0);
545     *resourceCount = mmap(NULL, sizeof(int),
546                 PROT_READ | PROT_WRITE, MAP_SHARED, *resFD, 0);
547 }
548
549 /**
550  * Validate command line parameters
551  * @param argc number of parameters
552  * @param argv command line parameters
553  */
554 void validateUse(int argc, char* argv[])
555 {
556     // Ensure correct number of command line parameters
557     if (argc != 3)
558     {
559         printf("Ensure there are the correct number of parameters\n");
560         exit(1);
561     }
562
563     // Ensure maxDelay is positive
564     if ( atoi(argv[2]) < 0)
565     {
566         printf("The maxDelay must be non-negative\n");
567         exit(1);
568     }
569 }
570
571 /**
572  * Close and destroy, semaphores and shared memory constructs
573  * @param buff1Ptr      Pointer to buffer1 in shared memory
574  * @param buff2Ptr      Pointer to buffer2 in shared memory
575  * @param countPtr      Pointer to counter in shared memory
576  * @param semaphores    Array of semaphores
577  * @param region        Array of region structs
578  * @param resourceCount Pointer to resourceCount in shared memory
579  * @param buff1FD       File descriptor for buffer1
580  * @param buff2FD       File descriptor for buffer2
581  * @param counterFD     File descriptor for counter
582  * @param semFD         File descriptor for semaphores
583  * @param regionFD      File descriptor for regions
584  * @param resFD         File descriptor for resourceCount
585  */
586 void cleanMemory(int (**buff1Ptr)[NINE][NINE], int **buff2Ptr, int** countPtr,
587                          sem_t **semaphores, Region **region,
588                             int** resourceCount, int buff1FD, int buff2FD,
589                                 int counterFD, int semFD, int regionFD,
590                                     int resFD )
591 {
592     // Close semaphores
593     sem_close(&((*semaphores)[0]));
594     sem_close(&((*semaphores)[1]));
595
596     // Destroy semaphores
597     sem_destroy(&((*semaphores)[0]));
598     sem_destroy(&((*semaphores)[1]));
599
600     // Clean up shared memory
601     shm_unlink("buffer1");
602     shm_unlink("buffer2");
603     shm_unlink("counter");
604     shm_unlink("semaphores");
605     shm_unlink("region");
606     shm_unlink("resources");
```

```
607
608     // Close file descriptors
609     close(buff1FD);
610     close(buff2FD);
611     close(counterFD);
612     close(semFD);
613     close(regionFD);
614     close(resFD);
615
616     // Unmap memory
617     munmap(*buff1Ptr, sizeof(int)*NINE*NINE);
618     munmap(*buff2Ptr, sizeof(int)*NUMPROCESSES);
619     munmap(*countPtr, sizeof(int));
620     munmap(*semaphores, sizeof(sem_t)*2);
621     munmap(*region, sizeof(Region)*NUMPROCESSES);
622     munmap(*resourceCount, sizeof(int));
623
624     // Unlink shared memory constructs
625     shm_unlink("buffer1");
626     shm_unlink("buffer2");
627     shm_unlink("counter");
628     shm_unlink("semaphores");
629     shm_unlink("region");
630     shm_unlink("resources");
631
632 }
```

5.2 Processes: mssv.h

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <fcntl.h>
#include <semaphore.h>
#include <sys/mman.h>
#include <unistd.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>
#include <signal.h>
#include <string.h>
#include <time.h>

#define NINE 9
#define SUB 3
#define NUMPROCESSES 11
#define FALSE 0
#define TRUE !FALSE

typedef enum {ROW, COL, SUB_REGION} Region_Type;

typedef struct
{
    Region_Type type;
    int positionX;
    pid_t pid;
    int valid;
} Region;


void readFile(char* inputFile, int rows, int cols, int (*buffer)[rows][cols]);
void writeFile(Region* region, char* format);
void resetArray(int numbers[]);
int checkValid(int numbers[]);
void parentManager(Region *region, sem_t *semaphores, int* countPtr,
                        int* resourceCount );
void childManager(Region *region, sem_t *semaphores,
                    int (*buff1Ptr)[NINE][NINE], int *buff2Ptr, int* countPtr,
                        int* resourceCount, int processNum, int *numbers,
                            int maxDelay );
void initMemory( int* buff1FD, int* buff2FD, int* counterFD, int* semFD,
                    int* regionFD, int* resFD);
void mapMemory(int* buff1FD, int* buff2FD, int* counterFD, int* semFD,
                    int* regionFD, int* resFD, int (**buff1Ptr)[NINE][NINE],
                        int (**buff2Ptr), int** countPtr, sem_t** semaphores,
                            Region** region, int** resourceCount);
void validateUse(int argc, char* argv[]);
void cleanMemory(int (**buff1Ptr)[NINE][NINE], int **buff2Ptr, int** countPtr,
                    sem_t **semaphores, Region **region, int** resourceCount,
                         int buff1FD, int buff2FD, int counterFD, int semFD,
                            int regionFD, int resFD);
```

5.3 Processes: Makefile

```
# Makefile For Sudoku Solution Validator
# COMP2006 Assignment
# Last Modified: 12/04/17
# Jordan Yeo - 17727626

# MAKE VARIABLES
EXEC1 = mssv
OBJ1 = mssv.o
CFLAGS = -std=c99 -pthread -D _XOPEN_SOURCE=500 -lrt
CC = gcc
INPUT = ../testFiles/specTest.txt
DELAY = 1

# RULES + DEPENDENCIES
$(EXEC1) : $(OBJ1)
  $(CC) $(OBJ1) -o $(EXEC1) $(CFLAGS)

mssv.o : mssv.c mssv.h #fileIO.h
  $(CC) -c mssv.c $(CFLAGS)

clean:
  rm -f $(EXEC1) $(OBJ1) logfile

run:
  ./$(EXEC1) $(INPUT) $(DELAY)
```

5.4 Threads: mssv.c

```c
#include "mssv.h"

pthread_mutex_t mutex; // Mutex
pthread_cond_t use;  // condition for if the global variable is in use

int **buff1, *buff2, *counter, maxDelay, inUse;
Region *regions;

int main (int argc, char* argv[])
{
    // Validate command line parameters
    validateUse(argc, argv);

    // Rename command line parameters
    char* inputFile = argv[1];
    maxDelay = atoi(argv[2]);

    // Variables
    pthread_t threads[11];

    // Generate random maxDelay
    srand((unsigned) time(NULL));
    maxDelay = rand() % maxDelay;

    // Allocate  memory
    initMemory( &buff1, &buff2, &counter, &regions);

    *counter = 0;
    // Read input file
    readFile(inputFile, NINE, NINE, &buff1);

    // Initialise mutex and condition
    pthread_mutex_init(&mutex, NULL);
    pthread_cond_init(&use, NULL);
    inUse = 0;

    // Create threads
    for(int i = 0; i < NUMTHREADS; i++)
    {
        if ( i < NINE) // Initialise region struct for row threads
        {
            regions[i].type = ROW;
        }
        else if( i == NINE) // Initialise region struct for columns thread
        {
            regions[i].type = COL;
        }
        else // Initialise region struct for sub-grids thread
        {
            regions[i].type = SUB_GRID;
        }

        regions[i].position = i;
        resetArray(regions[i].numbers);
        // Create thread
        pthread_create(&(threads[i]), NULL, childManager, &(regions[i]));
        inUse++;
    }

    parentManager(); // Parent logic

    cleanMemory(); // Clean up malloc'd memory

}

/*============================================================================*/
```

```
67
68 /**
69  * Initalise memory constructs
70  * @param buff1    buffer1 2D array
71  * @param buff2    buffer2 1D array
72  * @param counter  counter variable
73  * @param regions  Region struct 1D array
74  */
75 void initMemory(int*** buff1, int** buff2, int** counter, Region** regions)
76 {
77     // Initialise
78     *buff1 = (int**) malloc(sizeof(int*)* NINE);
79     for (int i = 0; i < NINE; i++)
80     {
81         (*buff1)[i] = (int*) malloc(sizeof(int)* NINE);
82     }
83     *buff2 = (int*) malloc(sizeof(int)* NUMTHREADS);
84     *counter = (int*) malloc(sizeof(int));
85     *regions = (Region*) malloc(sizeof(Region)* NUMTHREADS);
86 }
87
88
89 /**
90  * Free malloc'd memory and destroy mutex and conditions
91  */
92 void cleanMemory()
93 {
94     pthread_mutex_destroy(&mutex);
95     pthread_cond_destroy(&use);
96     for (int i = 0; i < NINE; i++)
97     {
98         free(buff1[i]);
99     }
100    free(buff1);
101    free(buff2);
102    free(counter);
103    free(regions);
104 }
105
106
107 /**
108  * Read the contents of the input file passed as a command line argument
109  * @param inputFile File to be read
110  * @param rows      Number of rows in matrix
111  * @param cols      Number of columns in matrix
112  * @param buffer    Matrix to store contents of input file
113  */
114 void readFile(char* inputFile, int rows, int cols, int***buffer )
115 {
116     FILE* inStrm;
117     int i, j;
118
119     inStrm = fopen(inputFile, "r"); // Open file for reading
120
121     if (inStrm == NULL) // Check file opened correctly
122     {
123         perror("Error opening file for reading\n");
124         exit(1);
125     }
126
127     // Store contents of file in 2D array
128     for( i = 0; i < rows; i++ )
129     {
130         for ( j = 0; j < cols; j++ )
131         {
132             fscanf( inStrm, "%d", &(*(buffer))[i][j] );
133         }
134     }
```

```
135     fclose(inStrm); // Close file
136 }
137
138 /**
139  * Write the invalid regions to log file
140  * @param region Sub region
141  * @param format String to be written
142  */
143 void writeFile(Region* region, char* format)
144 {
145     char* filename = "logfile";
146     FILE* outFile;
147     int val;
148
149     outFile = fopen(filename, "a"); // Open file for appending
150     if (outFile == NULL) // Check file opened correctly
151     {
152         perror("Error opening file for writing\n");
153         exit(1);
154     }
155
156     fprintf(outFile, "thread ID-%d: %s",region->tid, format);
157
158     fclose(outFile); // Close file
159 }
160
161 /**
162  * Set each index to zero
163  * @param numbers Array to be reset
164  */
165 void resetArray(int numbers[])
166 {
167     for (int i = 0; i < NINE; i++)
168     {
169         numbers[i] = 0;
170     }
171 }
172
173 /**
174  * Check if the contents of the array has any value other than one
175  * @param  numbers Array to be checked
176  * @return         Status of array being valid or not
177  */
178 int checkValid(int numbers[])
179 {
180     for (int j = 0; j < NINE; j++)
181     {
182         if ( numbers[j] != 1)
183         {
184             return FALSE;
185         }
186     }
187
188     return TRUE;
189 }
190
191 /**
192  * Handles the routine for the parent. Outputs the result to the screen
193  * @param threads ID of child threads
194  */
195 void parentManager()
196 {
197     char *type, *message;
198     int done = FALSE;
199     int position;
200
201     pthread_mutex_lock(&mutex); // Lock mutex
202     while ( inUse > 0 ) // Wait while children are executing
```

```
203     {
204         pthread_cond_wait(&use, &mutex);
205     }
206 
207     pthread_cond_signal(&use);
208     pthread_mutex_unlock(&mutex); // Unlock mutex
209 
210     for(int ii = 0; ii < NUMTHREADS; ii++)
211     {
212         pthread_mutex_lock(&mutex); // Lock mutex
213 
214         if (regions[ii].type == ROW)
215         {
216             type = "row";
217             position = regions[ii].position;
218             if ( regions[ii].valid == TRUE)
219             {
220                 printf("Validation result from thread ID-%u: %s %d is valid\n",
221                                         regions[ii].tid,type, position+1);
222             }
223             else
224             {
225                 printf("Validation result from thread ID-%u: %s %d is invalid\n",
226                           regions[ii].tid, type, position+1);
227             }
228         }
229         else if (regions[ii].type == COL)
230         {
231             type = "column";
232             printf("Validation result from thread ID-%u: %d out of 9 columns are valid\n"
233                     ,regions[ii].tid, regions[ii].count);
234         }
235         else
236         {
237             type = "sub-grid";
238             printf("Validation result from thread ID-%u: %d out of 9 sub-grids are valid\n"
239                     ,regions[ii].tid, regions[ii].count);
240         }
241 
242         pthread_mutex_unlock(&mutex);
243     }
244 
245 
246   if (*counter == 27)
247   {
248       message = "valid";
249   }
250   else
251   {
252       message = "invalid";
253   }
254 
255   printf("There are %d valid sub-grids, and thus the solution is %s\n", *counter, message);
256 
257 }
258 
259 /**
260  * Routine for child threads. Check the validity of sub region.
261  * @param args Void pointer to Region struct for the child
262  */
263 void* childManager(void* args )
264 {
265     char format[500];
266     int numValid;
267     Region* region = ((Region*)(args));
268     int threadNum = region->position;
269     int comma = 0;
270 
```

```
271     if( region->type == ROW ) // Check row in buffer1
272     {
273 
274         // Check rows
275         for (int i = 0; i < NINE; i++)
276         {
277             // Update numbers array
278             region->numbers[((buff1)[threadNum][i])-1]++;
279         }
280 
281         sleep(maxDelay); // Sleep
282         pthread_mutex_lock(&mutex); // Lock mutex
283 
284         // Update region struct
285         region->tid = pthread_self();
286         region->valid = checkValid(region->numbers);
287 
288         // Update buffer2
289         numValid = 0;
290         if (region->valid == TRUE)
291         {
292             numValid = 1;
293             region->count = numValid;
294         }
295         else // Write to log file
296         {
297             region->count = numValid;
298             sprintf(format, "row %d is invalid\n", threadNum+1);
299             writeFile((region), format);
300         }
301 
302         buff2[threadNum] = numValid; // Update buffer2
303 
304         *counter = *counter + numValid; // Update counter
305 
306         pthread_mutex_unlock(&mutex); // Unlock mutex
307 
308     }
309     else if( region->type == COL ) // Check all columns
310     {
311         sprintf(format, "column ");
312         int validCol = 0;
313         for ( int nn = 0; nn < NINE; nn++) // Iterate through each column
314         {
315             for(int ii = 0; ii < NINE; ii++) // Iterate through each row
316             {
317                 // Update numbers array
318                 region->numbers[((buff1)[ii][nn])-1]++;
319             }
320 
321             // Check if the column is valid
322             if ( checkValid( region->numbers) == TRUE )
323             {
324               validCol++;
325             }
326             else
327             {
328                 if (comma == 0)
329                 {
330                     comma = 1;
331                     sprintf(format + strlen(format), "%d", nn+1);
332                 }
333                 else
334                 {
335                     sprintf(format + strlen(format), ", %d ", nn+1);
336                 }
337             }
338             resetArray(region->numbers);
```

```
339         }
340
341         sleep(maxDelay);
342         if (validCol == 8)
343         {
344             sprintf(format + strlen(format), " is invalid\n");
345         }
346         else
347         {
348             sprintf(format + strlen(format), "are invalid\n");
349         }
350         pthread_mutex_lock(&mutex); // Lock mutex
351
352         // Update region struct
353         region->count = validCol;
354         region->tid = pthread_self();
355         if(validCol != 9)
356         {
357             writeFile((region), format);
358         }
359
360         numValid = region->count;
361
362         buff2[threadNum] = validCol; // Update buffer2
363
364         *counter = *counter + validCol; // Update counter
365
366         pthread_mutex_unlock(&mutex); // Unlock mutex
367     }
368     else if( region->type == SUB_GRID ) // Check all sub-grids
369     {
370         sprintf(format, "sub-grid ");
371
372         int validSub = 0;
373
374         // Iterate through each of the 9 3x3 sub-grid
375         for ( int jj = 0; jj < 3; jj++)
376         {
377             for (int kk = 0; kk < 3; kk++)
378             {
379                 for (int ll = jj*3; ll < jj*3+3; ll++)
380                 {
381                     for (int mm = kk*3; mm < kk*3+3; mm++)
382                     {
383                         // Update numbers array
384                         region->numbers[((buff1)[ll][mm])-1]++;
385                     }
386                 }
387
388                 if ( checkValid(region->numbers) == TRUE )
389                 {
390                     validSub++;
391                 }
392                 else // Update string for log file
393                 {
394                     if (comma == 0)
395                     {
396                         comma = 1;
397                         sprintf(format+strlen(format), "[%d..%d, %d..%d]",
398                                 jj*3+1, jj*3+3, kk*3+1, kk*3+3);
399                     }
400                     else
401                     {
402                         sprintf(format+strlen(format), ", [%d..%d, %d..%d]",
403                                 jj*3+1, jj*3+3, kk*3+1, kk*3+3);
404                     }
405                 }
406                 resetArray(region->numbers);
```

```
407             }
408
409         }
410
411         sleep(maxDelay);
412         if( validSub == 8)
413         {
414             sprintf(format+strlen(format), " is invalid\n");
415         }
416         else
417         {
418             sprintf(format+strlen(format), " are invalid\n");
419         }
420         pthread_mutex_lock(&mutex); // Lock mutex
421
422         // Update region struct
423         region->count= validSub;
424         region->tid = pthread_self();
425
426         if(validSub != 9)
427         {
428             writeFile((region), format);
429         }
430
431         buff2[threadNum] = validSub; // Update buffer2
432
433         *counter = *counter + validSub; // Update counter
434
435         pthread_mutex_unlock(&mutex); // Unlock mutex
436
437     }
438
439     // Child signals it is finished by incremented resourceCount
440     pthread_mutex_lock(&mutex); // Lock mutex
441
442     while( inUse == 0)
443     {
444         pthread_cond_wait(&use, &mutex);
445     }
446     inUse--; // Decrease count of child processes running
447     if (inUse == 0)
448     {
449         pthread_cond_signal(&use);
450     }
451     pthread_mutex_unlock(&mutex); // Unlock mutex
452     pthread_detach(pthread_self()); // Release resources
453
454 }
455
456 /**
457  * Validate command line parameters
458  * @param argc number of parameters
459  * @param argv command line parameters
460  */
461 void validateUse(int argc, char* argv[])
462 {
463     // Ensure correct number of command line parameters
464     if (argc != 3)
465     {
466         printf("Ensure there are the correct number of parameters\n");
467         exit(1);
468     }
469
470     // Ensure maxDelay is positive
471     if ( atoi(argv[2]) < 0)
472     {
473         printf("The maxDelay must be non-negative\n");
474         exit(1);
```

```
475         }
476 }
```

5.5 Threads: mssv.h

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <fcntl.h>
#include <semaphore.h>
#include <sys/mman.h>
#include <unistd.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>
#include <signal.h>
#include <string.h>
#include <time.h>
#include <pthread.h>

#define NINE 9
#define SUB 3
#define NUMTHREADS 11
#define FALSE 0
#define TRUE !FALSE

typedef enum {ROW, COL, SUB_GRID} Region_Type;

typedef struct
{
    Region_Type type;
    int position;
    pthread_t tid;
    int count;
    int valid;
    int numbers[NINE];

} Region;


void readFile(char* inputFile, int rows, int cols, int***buffer);
void writeFile(Region* region, char* format);
void resetArray(int numbers[]);
int checkValid(int numbers[]);
void parentManager(void);
void* childManager(void* args );
void initMemory( int*** buff1, int** buff2, int** counter, Region** regions);
void mapMemory(int* buff1FD, int* buff2FD, int* counterFD, int* semFD,
                    int* regionFD, int* resFD, int (**buff1Ptr)[NINE][NINE],
                        int (**buff2Ptr), int** countPtr, sem_t** semaphores,
                            Region** region, int** resourceCount);
void validateUse(int argc, char* argv[]);
void cleanMemory(void);
```

5.6 Threads: Makefile

```
# Makefile For Sudoku Solution Validator
# COMP2006 Assignment
# Last Modified: 12/04/17
# Jordan Yeo - 17727626

# MAKE VARIABLES
EXEC1 = mssv
OBJ1 = mssv.o
CFLAGS = -std=c99 -g -pthread -D _XOPEN_SOURCE=500 -lrt
CC = gcc
INPUT = ../testFiles/specTest.txt
DELAY = 10


# RULES + DEPENDENCIES
$(EXEC1) : $(OBJ1)
  $(CC) $(OBJ1) -o $(EXEC1) $(CFLAGS)

mssv.o : mssv.c mssv.h #fileIO.h
  $(CC) -c mssv.c $(CFLAGS)

#fileIO.o : fileIO.c fileIO.h
# $(CC) -c fileIO.c $(CFLAGS)

clean:
  rm -f $(EXEC1) $(OBJ1) logfile

run:
  ./$(EXEC1) $(INPUT) $(DELAY)
```